MegTaiChi: Dynamic Tensor-based Memory Management Optimization for DNN Training

Zhongzhe Hu
Institute of Computing Technology, Chinese Academy of Sciences
huzhongzhe@ncic.ac.cn

Junmin Xiao*
Institute of Computing Technology, Chinese Academy of Sciences
xiaojunmin@ict.ac.cn

Zheye Deng
Megvii Technology Co.,Ltd.
dengzheye@megvii.com

Mingyi Li
Institute of Computing Technology, Chinese Academy of Sciences
limingyi@ncic.ac.cn

Kewei Zhang
Institute of Computing Technology, Chinese Academy of Sciences
zhangkewei@ncic.ac.cn

Xiaoyang Zhang
Institute of Computing Technology, Chinese Academy of Sciences
zhangxiaoyang@ncic.ac.cn

Ke Meng
Alibaba Group
septic.mk@gmail.com

Ninghui Sun
Institute of Computing Technology, Chinese Academy of Sciences
snh@ict.ac.cn

Guangming Tan
Institute of Computing Technology, Chinese Academy of Sciences
tgm@ict.ac.cn

ABSTRACT

In real applications, it is common to train deep neural networks (DNNs) on modest clusters. With the continuous increase of model size and batch size, the training of DNNs becomes challenging under restricted memory budget. The tensor partition and tensor rematerialization are two major memory optimization techniques to enable larger model size and batch size within the limited-memory constrain. However, the related algorithms failed to fully extract the memory reduction opportunity, because they ignored the invariable characteristics of dynamic computational graphs and the variation among the same size tensors at different memory locations. In this work, we propose MegTaiChi, a dynamic tensor-based memory management optimization module for the DNN training, which first achieves an efficient coordination of tensor partition and tensor rematerialization. The key feature of MegTaiChi is that it makes memory management decisions based on dynamic tensor access pattern tracked at runtime. This design is motivated by the observation that the access pattern to tensors is regular during training iterations. Based on the identified patterns, MegTaiChi exploits the total memory optimization space and achieves the heuristic, adaptive and fine-grained memory management. The experimental results show, MegTaiChi can reduce the memory footprint by up to 11% for ResNet-50 and 10.5% for GL-base compared with DTR. For the training of 6 representative DNNs, MegTaiChi outperforms MegEngine and Sublinear by 5× and 2.4× of the maximum batch sizes. Compared with FlexFlow, Gshard and ZeRo-3, MegTaiChi achieves 1.2×, 1.8× and 1.5× performance speedups respectively on average. For the million-scale face recognition application, MegTaiChi achieves 1.8× speedup compared with the optimal empirical parallelism strategy on 256 GPUs.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Parallel algorithms**.

KEYWORDS

Dynamic datafow graph, Tensor partition, Tensor rematerialization

ACM Reference Format:
Zhongzhe Hu, Junmin Xiao*, Zheye Deng, Mingyi Li, Kewei Zhang, Xiaoyang Zhang, Ke Meng, Ninghui Sun, and Guangming Tan. 2022. MegTaiChi: Dynamic Tensor-based Memory Management Optimization for DNN Training. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524059.3532394

1 INTRODUCTION

The current popular deep learning frameworks support both declarative and imperative style programming, which are based on the static and dynamic computational graph(SCG and DCG) respectively. With the development of various models, the programming mode based on DCG is more convenient to deploy and debug a new model compared to SCG, which is flexible for the exploration of deep neural network (DNN) architectures.

As state-of-the-art deep learning models continue to grow, model-training within the constraints of on-device memory becomes increasingly challenging. Besides, the scarce on-device memory resource also limits the training with large batch sizes. Related works show that, the major memory consumption in the DNN training comes from storing model parameters and intermediate layer outputs which are generated during forward propagation and used again in the backward propagation. In current deep learning frameworks, intermediate outputs are usually maintained in on-device

* Corresponding author.

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06.
https://doi.org/10.1145/3524059.3532394

memory until they are no longer needed, resulting in high memory consumption that prevents model size and batch sizes.

To tackle the challenges above, two major techniques are developed: *tensor partition* and *tensor rematerialization*. The *tensor partition* is to split model parameters of different layers into partitions and place each partition on a separate machine where the training for the partition takes place [8, 9, 17, 25, 27, 32]. The *tensor rematerialization* focused on reducing the on-device memory footprint on a single machine. This technique is based on the design principle of releasing the memory for intermediate tensors in the forward propagation and regenerating the released tensors in backward propagation if necessary[6]. In general, the techniques both reduce the entry barrier of processing large models and enable large batch sizes without model accuracy loss. Additionally, there are alternative ways to reduce memory consumption, such as using lower-precision computations [16] and compressing model parameters via quantization and sparsification [4, 26]. However, these approaches affect model accuracy and require heavy hyperparameter tuning, which are not considered in our work.

In real applications, it is common to train DNNs with large model size or batch sizes on clusters of modest size, such as training BERT model in a private cloud platform with few GPUs [23]. In this scenario, our work combines the tensor partition and rematerialization together for memory management on DCG. Although these two optimization techniques can both reduce memory footprint, they were developed separately [28, 29] based on different application requirements. In fact, the tensor partition was designed for training a large model on multiple machines, which aimed to achieve load balance to maximize parallelism and locality to minimize network communication. However, the tensor rematerialization was proposed to enable the training with on-device memory constraints on a single machine, which aimed to save the on-device memory space by relying on secondary storage or external computation. It seems natural to combine them to improve the performance of model-training on modest clusters. However, three complications make the combination non-trivial. (1) Since recent tensor partition plans[15, 19, 23, 28, 29, 33] are usually defined before the execution and fixed during the training, these plans don't take into account the runtime information which essentially determines the tensor rematerialization. Hence, the first issue is how to achieve dynamic adjustment of tensor partition during the runtime. (2) The second issue is how to determine which history tensors should be evicted and later regenerated after the current tensor partition plan is generated. [6, 13, 14, 18, 22]. (3) As both the tensor partition and rematerialization plans control the life span of tensors in on-device memory, it is necessary to consider the optimization of memory space assignment. The third issue is how to assign the memory space for each tensor to improve the memory usage.

Our work is based on two observations. (1) Similar to reuse and locality analysis of memory accesses for traditional programs, tensor accesses in deep learning training also exhibit data reuse and fixed patterns. (2) There are some invariable characteristics in DCGs which are built dynamically across millions of iterative steps. These two observations give the opportunity to combine tensor partition and rematerialization. Firstly, abstracting and concluding efficient tensor partition patterns could achieve the immediate generation of tensor partition plans during the runtime. Secondly, tracking the runtime information about tensor accesses could guide releasing and regenerating tensors. Thirdly, exploiting the invariable characteristics of DCGs could make it possible to arrange the spatial position of tensors in memory for global planning memory usage.

In this work, we propose a dynamic computational graph based memory management module called MegTaiChi, which successfully combines the tensor partition and rematerialization techniques together, and enables dynamic memory management at operator and tensor granularity. MegTaiChi is *heuristic*, *adaptive* and *fine-grained*. (1) *Heuristic*: the tensor partition is generated dynamically based on pattern-driven models. (2) *Adaptive*: the tensor rematerialization is adjusted automatically according to the current memory usage. (3) *Fine-grained*: the memory allocation is precise based on the memory rearrangement of tensor positions.

We summarize our contributions as follows.

- Propose a dynamic tensor-based memory management optimization module MegTaiChi for DNNs training, which first achieves an efficient coordination of tensor partition and tensor rematerialization techniques.
- Develop a dynamic tensor partition strategy which achieves dynamic adjustment during the model training process.
- Design a dynamic tensor management strategy which tradeoffs swapping and recomputing mechanisms on memory management.
- Propose an optimal memory allocation scheme to further alleviate the memory fragmentation problem by exploiting the invariable characteristics of DCGs.
- Evaluate and analyze the performance of the proposed optimizations, which proves that MegTaiChi supports large-scale DNN training well.

2 BACKGROUND

2.1 Computational Graph

In major deep learning frameworks, computational graphs are applied to express the training process, where operators or functions are abstracted as computational nodes, and data dependency is represented by edges. A computational graph is usually classified into two distinct types: dynamic and static, which respectively correspond to imperative and declarative programming style.

Imperative style framework such as PyTorch[21] defines and performs DCG during execution, which is termed as define-by-run [3, 18]. When a tensor is defined in a DCG, its value has been determined. This programming mode is convenient for deploying and debugging models, and flexible for representing the training process of various DNNs. Hence, DCG is particularly popular in academic community.

Declarative style framework such as Lazy-TensorFlow[2] defines SCG before performing it, which is termed as define-and-run. When SCG is constructed, actual computations do not take place. All operations can be scheduled before execution. Hence, the training on SCG is easy to achieve higher performance and lower memory usage compared with DCG.

2.2 Tensor Partition

For training a large model on multiple machines, tensor partition is to split model parameters of intermediate layers into multiple smaller tensors, and distribute them to different machines [8, 9, 17,

25, 27, 32], which can lower the per-machine memory footprint. When the tensor partition is across k machines, each machine roughly consumes $1/k$ of the total memory required to run the computation on one machine. Besides, partitioning also has the important benefit of performance speedup via parallel execution. Our goal is to automatically partition the tensors and parallelize the operators on DCG to enable the training with large batch sizes or models.

2.3 Tensor Rematerialization

Tensor rematerialization is a technique to enable the training with large models or batch sizes that exceeds limited memory on a single machine without modifying models, which usually involves two general methods: *swapping* and *recomputing* [6, 13, 14, 18, 22]. Both methods are based on the design principle of releasing some parts of intermediate tensors in the forward propagation and regenerating them in backward propagation. They differ in how regeneration is performed. Specifically, The *swapping* method leverages the host memory as an external memory and copies the intermediate data back and forth between host and on-device memory asynchronously, while the *recomputing* method releases some intermediate tensors directly from on-device memory and regenerates them again by replaying the forward computation if necessary. In general, both techniques above do not affect training accuracy. In this paper, we only consider DCG. Figure 1(a) shows an example of rematerialization on DCG. When $OP4$ needs to calculate a tensor f, the input tensor e is not in memory (it has been evicted). $OP3$ needs to be executed to regenerate tensor e firstly. But, OOM is triggered at this time. Evicting tensors from the historical tensor set is necessary. Swapping tensor a and b to host confirms $OP3$ to generate tensor e. For executing $OP4$, OOM occurs again. Hence, it is necessary to evict tensor c from the tensors remaining in memory for successfully generating tensor f.

2.4 Fragmentation and Defragmentation

From Figure 1(a), it is clear that the tensor rematerialization needs to dynamically free and allocate memory frequently during this process, which will lead to serious *memory fragmentation*. As shown in Figure 1(b), when the small memory chunk for the tensor b is released in time T_5, the large memory request for e cannot reuse the memory chunk for b. At this time, it is necessary to defragment the memory, but the *defragmentation* operation requires the help of the host memory, which will seriously affect performance.

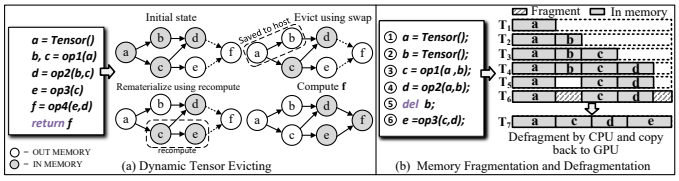


Figure 1: (a) DTE applys on an instance. (b) DTE brings memory fragmentation problem.

3 APPROACH AND CHALLENGES

3.1 Opportunity

The design of MegTaiChi is based on two key observations. First, the memory management on DCG seems difficult to be optimized since the computational graph is unknown before the execution, but the processing procedures are based on tensor operations. Similar to reuse and locality analysis of memory accesses for traditional programs, tensor accesses in deep learning training also exhibit data reuse and certain access patterns. Thus, we believe that dynamically tracking fine-grained tensor accesses provides the foundation for effective memory management optimizations on DCG.

Second, the properties of DNN structure ensure the effectiveness of our approach. The training process consists of millions of iterative steps. Across iterations, the tensor accesses have some repeated and fixed access patterns, and DCGs have some invariable characteristics even though they are built dynamically. This means that analyzing the tensor access patterns and the invariable characteristics of DCG can easily reveal the memory optimization opportunities with concrete guidance, e.g., how to partition tensors.

3.2 Our Approach

As Figure 2 shown, MegTaiChi is a virtual machine (VM) module for memory manage optimization on DCG by controlling tensor access behaviors with basic primitive operations, e.g., GetValue, DeLete, SwapIn, SwapOut, Drop, etc. MegTaiChi can manage the dynamic memory access at the tensor granularity from both the spatial and temporal localities. First, for the spatial locality, the dynamic tensor partition is proposed to determine whether or how to partition each tenor, by balancing the execution cost and memory cost. Second, for the temporal locality, the dynamic tensor evicting is designed to determine whether or when to intercept tensor allocations, accesses, and deallocations by avoiding the release of high frequency used data and balancing the tensor rematerialization time and memory footprint. Third, together the spatial locality with temporal locality, the near-optimal memory allocation is developed to determine the exact memory address for each tensor to make full use of all the device memory of whole system by capturing and using the important invariable characteristics of DCG and avoiding the generation of memory fragments.

Dynamic Tensor Partition (DTP) is a heuristic design for partitioning tensors. When VM instructions are generated for an operator OP to be performed, MegTaiChi needs to determine how to execute the OP. According to the meta information of the inputs and outputs, e.g., the shape of input tensors, DTP would generate a partition schedule for all the tensors which take part in the OP, which leads to an execution plan for the OP. As shown in Panel ❶ of Figure 2, during the instruction generation for executing OP_{i+1}, MegTaiChi applies DTP to infer the tensor partition plan of OP_{i+1} heuristically, which may need to change the partition dimension of outputs of OP_i and the partition dimension of weights before executing OP_{i+1}. Consequently, the relevant communication instruction will be generated and inserted in front of the computation instruction of OP_{i+1}. It is worth mentioning that DTP successfully expands the conventional tensor partition technique from the training on SCGs to DCGs.

Dynamic Tensor Evicting (DTE) is an adaptive strategy for releasing tensors during the runtime. When the VM instructions are executed, MegTaiChi tracks the execution sequence of OPs, and records the execution information, e.g., the history of memory allocation and deallocation, memory access time, each tensor's ancestry and other metadata. Once the out of memory (OOM) occurs, DTE would determine which tensors should be released and whether

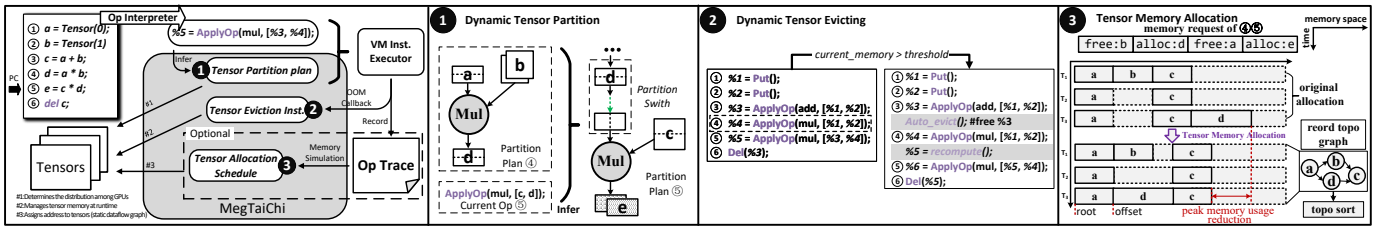


Figure 2: MegTaiChi Overview.

to choose swapping or recomputing for the selected tensors. As shown in Panel ❷ of Figure 2, before the execution of the original *OP4*: $a * b = c$, the *current_memory* > *thre_shold* appears, which indicates that OOM happens. Immediately, the *Autoe_vict()* instruction is generated inserted into the VM instruction queue, which releases some tensors from memory based on DTE. In addition, if the memory access failure occurs, the needed tensors missing in memory would be regenerated by the corresponding rematerialization methods which are marked at their evicting moment. It is worth mentioning that DTE fully exploiting the advantages of both the swapping and recomputing methods for memory management on DCGs.

Tensor Memory Allocation (TMA) is a fine-grained method for assigning memory address to all the tensors. If the architecture of DNN model is fixed during the training process, the DCG would be invariable even though it is generated dynamically. During the first five training iterations, MegTaiChi captures the important invariable characteristics of DCGs, e.g. the life span of intermediate tensors at a training iteration, by tracing the VM instruction sequence. Based on the invariable characteristics of DCGs, TMA can establish a fine-grained memory allocation plan which would be adopted in the following epochs until the end of training process. As shown in Panel ❸ of Figure 2, using the memory requirement information, TMA simulates the memory allocation process and adjusts the starting address of some tensors in the simulation, which leads to a topological map specifying the dependency relationship between all the tensors on time and space dimension degrees. Based on the topological map, TMA can generate a near-optimal memory allocation plan by leveraging a sort algorithm. It is worth mentioning that TMA successfully reduce the memory peak value and avoids the generation of memory fragments.

The three components are coordinated by a VM interpreter. During the iteration, the interpreter invokes DTP inference and executes the corresponding instructions for tensor partition. If OOM is triggered in execution, the interpreter would generate a DTE instruction for acquiring enough memory dynamically. After the memory access sequence is cached, the interpreter could apply TMA to plan the memory allocation.

3.3 Challenges

3.3.1 How to Partition Tensors on Multiple Machines. In major deep learning frameworks, the tensor partition plans are usually defined before the execution and fixed during the training. Hence, the tensor partition design is difficult to take into account the runtime information, while the runtime information essentially determines the tensor rematerialization plan [18]. To change this situation, this work trends to develop a dynamic tensor partition strategy

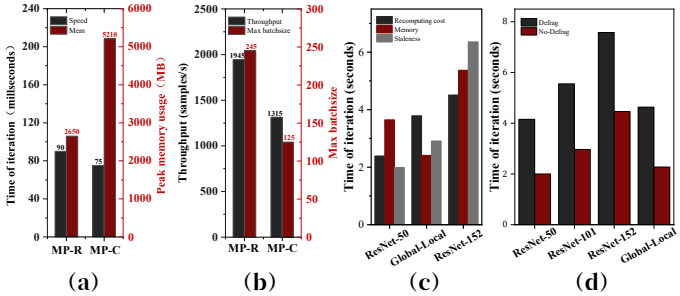


Figure 3: Panel (a) and Panel (b) show the performance of ResNet-18 training. MP-R and MP-C represent model parallelisms based on row and column partitions respectively. Pane (c) shows the evaluation of different tensor evicting schemes based on only one factor. Panel (d) shows the effect of defragmentation procedures to the performance.

which can be adjusted according to the current memory usage during the execution. Underlying the design of dynamic tensor partition, our fundamental assumption is that data can be streamed from a remote device at the same rate as from a local device. This assumption holds true for clusters of modest size, which is the environment targeted by MegTaiChi. Recent work on datacenter networks suggests that this assumption also holds on a larger scale [10, 20]. Under this assumption, the network is never the bottleneck. In this case, the memory usage becomes improtant, which affects the final performance. As Figure 3(a)-(b) shown, MP-R is slightly slower than MP-C, but MP-R consumes smaller memory, which could lead to higher throughput and final performance. Hence, the design of DTP needs to balance the execution time and memory usage.

3.3.2 How to Evict Tensors from On-device Memory. For the tensor rematerialization, the swapping and recomputing methods have been widely used. In this work, DTE tries to combine the swapping and recomputing together. For each tensor, many factors would affect the prediction whether the resident tensor may be least valuable and should be evicted, such as staleness (the time since last access the tensor), memory (the size of the tensor), and the recomputing cost (time required to compute the tensor from its parent tensors). Figure 3(c) shows the evaluation of different tensor evicting schemes each of which is heuristic based on only one factor. It is clear that these factors alternately become the primary in different cases. Hence, the design of DTE should investigate the whole effect of major factors. Besides, to reduce the memory fragments, DTE should take into account the variation among the same size tensors

stored at different spatial positions of memory, which is ignored by related works [18, 18, 22].

3.3.3 How to Allocate Memory Space for All the Tensors. The tensor partition and rematerialization usually involve repeated allocating and releasing data, which would result in a mass of memory fragments and affect the memory usage. For example, although the family of ResNet models can be stored in the on-device memory of a single GPU, the training with large batch sizes still needs defragmentation procedures to keep the workflow continuing. However, the defragmentation procedure must affect the execution overhead as Figure 3(d) shown. In order to make full use of the memory space, this work would investigate the memory allocation for each tensor. The design of TMA is motivated by an attractive observation that different training iterations in the same stage share a large amount of or even all of the tensors, which means that the training has extremely good inter-task data locality. Inspired by this fact, MegTaiChi would try to avoid the generation of memory fragments and reduce the peak memory usage by leveraging the traced memory access sequence to guide the fine-grained memory allocation.

4 IMPLEMENTATION OF MEGTAICHI

4.1 Dynamic Tensor Partition

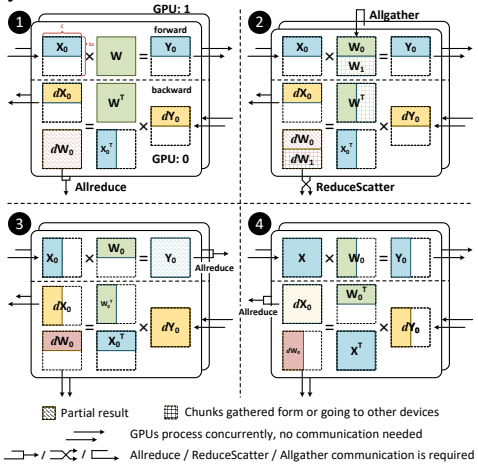


Figure 4: Four Efficient Tensor Partition Patterns.

Formal Representation: In the neural network training, if the input and output variables of each operator (or layer) is written in the matrix form, the forward phase is a sequential combination of affine transformation $\mathbf{Y} = \mathbf{X} \times \mathbf{W}$, followed by nonlinear transforms $\mathbf{X}' = g(\mathbf{Y})$ in which g is a nonlinear correction function (such as ReLU or Sigmoid function), and each column of $\mathbf{X}$ holds input activation tensors for one sample, and similarly each column of $\mathbf{Y}$ holds output activation tensors for one sample. $\mathbf{X}$ and $\mathbf{Y}$ have the same shape. The matrix $\mathbf{W}$ holds the weights of the neural network between the current layer and the next one. Denote c as the current layer's number of neurons (or the tensor dimension), and bs represents the batch size. Similarly, the backward propagation can also be written in matrix form as $d\mathbf{X} = \mathbf{W}^T \times d\mathbf{Y}$. Here, $d\mathbf{X}$ and $d\mathbf{Y}$ are the gradients of the loss function with respect to $\mathbf{X}$ and $\mathbf{Y}$ respectively. Meanwhile, the gradient of the loss function with respect to model weights $\mathbf{W}$ is calculated as $d\mathbf{W} = \mathbf{X}^T \times d\mathbf{Y}$.

Table 1: The Inter-operator Collective Communication.

	P1	P2	P3	P4
P1	None	None	Alltoall	Allgather
P2	None	None	Alltoall	Allgather
P3	ReduceScatter	RedcueScatter	ReduceScatter	Allredcue
P4	Alltoall	Alltoall	None	Allgather

Consequently, the tensor partition on each layer involves the three matrix multiplications above.

Pattern Abstraction: In this work, we deeply investigate all commonly used tensor partitions, and abstract four efficient patterns, as shown in Figure 4. ***Pattern 1*** partitions $\mathbf{X}$ and $\mathbf{Y}$ respectively along the row dimension, which is usually called as data parallel partition. Under Pattern 1, the forward pass needs no communication between different devices, while the backward pass has to use Allreduce communication procedure to obtain the whole $d\mathbf{W}$. ***Pattern 2*** partitions $\mathbf{W}$, $\mathbf{X}$ and $\mathbf{Y}$ respectively along the row dimension. As different parts of $\mathbf{W}$ are distributed on multiple devices, the forward pass needs to collect them using Allgather procedure. During the backward pass, each device updates the parts of $\mathbf{W}$ independently, which involves ReduceScatter procedure. When the size of $\mathbf{W}$ is large, a considerable amount of memory can be saved for each device. ***Pattern 3*** partitions $\mathbf{W}$ along row, and partition $\mathbf{X}$ along column dimension. In the forward pass, each device needs Allreduce procedure to obtain the whole $\mathbf{Y}$. But, the backward pass does not need collective communication. ***Pattern 4*** partitions $\mathbf{W}$ and $\mathbf{Y}$ respectively along column dimension. Under Pattern 4, the forward pass involves no communication, while the backward pass has to collect different parts of $d\mathbf{W}$ to devices using Allreduce procedure.

In Addition, Figure 4 summarizes the required intra-operator communication procedures for four patterns in forward, backward and gradient update processes (Parallel arrow indicates that no communication is needed). If two adjacent operators use different tensor partition patterns, the additional communication is needed to switch partition patterns from Pattern k_1 for the current operator to Pattern k_2 in the next operator during the forward pass ($1 \leq k_1, k_2 \leq 4$). Besides, the communication procedures needed for pattern switch are shown in Table 1.

Dynamic Tensor Partition: To achieve high throughput in the training process, it is improtant to balance the execution cost and the memory consumption that would affect the concurrency, i.e., batch size. For the i-th operator o_i, we denote p_i^k as the partition configuration which represents using Pattern k to partition the data of o_i. With Pattern k, the total execution time $t(o_i, p_i^k, bs)$ and memory cost $m(o_i, p_i^k, bs)$ of the i-th operator can be represented as follows

$$t\left(o_i, p_i^k, bs\right) = t_c\left(o_i, p_i^k, bs\right) + t_{intra}\left(o_i, p_i^k, bs\right) + t_{inter}\left(p_i^k, p_{i-1}^{k'}, bs\right),$$

and

$$m\left(o_i, p_i^k, bs\right) = m_p\left(o_i, p_i^k, bs\right) + m_t\left(o_i, p_i^k, bs\right) + m_{inter}\left(p_i^k, p_{i-1}^{k'}, bs\right),$$

where t_c represents the time taken to conduct the computation defined by operators. t_{intra} and t_{inter} are the intra and inter operator communication costs respectively. m_p is the memory for storing the (partitioned) model parameter, and m_t is the memory

for intermediate tensors, and m_{inter} is the memory for intermediate tensors generated by the pattern switching from Pattern k' for o_{i-1} to Pattern k for o_i. Note that all the time and memory costs at the right-hand side of two equations above can be measured and recorded during the training, and they are the functions of batch size bs due to the batch size determining the size of intermediate tensors.

DTP is a heuristics that selects Pattern k_i^* for the tensor partition of o_i by solving an optimization problem

$$k_i^* = \underset{k\in\{1,2,3,4\}}{\arg\min}\ \lambda \cdot m\left(o_i, p_i^k, bs\right) + (1-\lambda)\cdot t\left(o_i, p_i^k, bs\right), \quad (1)$$

where λ is a constant satisfying $0 \le \lambda \le 1$. Note that λ can be adjusted by users.

4.2 Dynamic Tensor Evicting

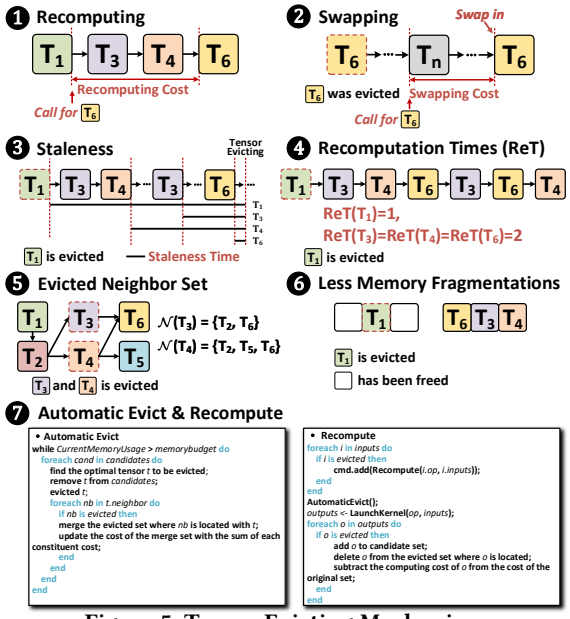


Figure 5: Tensor Evicting Mechanism.

As mentioned earlier, it is necessary to select and evict some tensors when there is not enough free memory for the current operator to execute. For tensor evicting, the major technique is the tensor rematerialization which includes two different methods: recomputing and swapping (as shown in Panel ❶ and ❷ of Figure 5). Recent works just focused on the memory management on a single device, and ignored the serious memory fragmentation caused by evicting tensors repeatedly from the on-device memory. The design of DTE is to solve two issues: (1) how to manage the on-device memory on multiple devices; (2) how to determine which tensors should be evicted from the on-device memory by the recomputing method or the swapping to avoid the generation of memory fragments.

Memory Management Mechanism: The training on multiple devices involves both computation and communication. The gradients of different layers' tensors commonly arriving at the current GPU out of order, makes it difficult to predict the consumption of memory. To develop DTE for the training on multiple devices, we propose a memory regional management mechanism. (1) Firstly, we divide the on-device memory of each device into two regions, i.e. $m = m_1 + m_2$. The first region is reserved for the communication between devices to store the gradients, while the other one is used for the computation process on the current device. (2) Next, during the training, the received gradients are placed on the first memory region which is set large enough to store the gradients of all the parameters and $1/p$ of the largest layer's input and output tensors across p devices. (3) Finally, we define a *memory threshold* m^* for the usage of the second region m_2. If the memory consumption of the second region is larger than m^*, we would select and evict some tensors until the consumed memory is less than m^*, which means that the memory space with size of $m_2 - m^*$ is reserved for the execution of current operator.

Tensor Evicting Mechanism: In each iteration, the metadata for each tensor t can be tracked, such as ***Staleness*** $s(t)$ (time since last access), ***Memory*** $m(t)$ (tensor size), ***Cost*** $c(t)$ (time required to calculate t from its parent tensors based on its computational path) ***Recomputing times*** $ret(t)$ (tensor recomputing times). In the design of DTE, we tend to evict the tensor t which is stalest (to prioritize the evicting of tensor that is least recently used, see Panel ❸ of Figure 5), largest (to save as much memory as possible), cheapest (to minimize the additional recomputing cost), and least frequent (to avoid incessant tensor recomputing which may bring heavy overhead, see Panel ❹ of Figure 5).

We use M to denote the set that consists of all the current tensors stored in the on-device memory. For each tensor $t \in M$, we denote its *evicted neighbor set* as $\mathcal{N}(t)$ (refer to Panel ❺ of Figure 5), which is the set of evicted tensors that would either need to be recomputed to obtain t again or would need t to be resident to be recomputed. For a candidate tensor t to be evicted, its recomputing cost $\mathbb{C}_r(t)$ can be estimated by $\mathbb{C}_r(t) = c(t) + \sum_{\tau\in\mathcal{N}(t)} c(\tau)$, while its swapping cost $\mathbb{C}_s(t)$ can be quantified as the time for loading t from the out-device memory to on-device memory. Let $m(t)$ be the the size of t, and γ be the bandwidth between the out-device memory and on-device memory, we have $\mathbb{C}_s(t) = m(t)/\gamma$.

In order to decrease the on-device memory fragments generated by repeated evicting tensors, we prioritize the select and release of those tensors whose evicting would not generate new memory fragments. For instance, there are two tensors as candidates for evicting, and the candidate should acquire a relatively higher freeing priority if its left and right neighbors have been already freed from the memory. For a tensor t, we denote $M_{left}(t)$ and $M_{right}(t)$ as the sizes of t's left and right free memories respectively, and tend to evict the tensor t whose $m(t) + M_{left}(t) + M_{right}(t)$ is larger (Panel ❻ of Figure 5). As Panel ❼ of Figure 5 shown, DTE is to repeatedly evict tensors by minimizing the following cost until there is enough free memory for the execution of current operation,

$$\min_{t\in M} \frac{\min(\mathbb{C}_r(t), \mathbb{C}_s(t)) \cdot \beta^{ret(t)}}{\left(m(t) + M_{left}(t) + M_{right}(t)\right)\cdot s(t)}. \quad (2)$$

Specifically, β is chosen as an empirical value of 0.5. For the selected t to be evicted, we compare the recomputing cost $\mathbb{C}_r(t)$ and swapping cost $\mathbb{C}_s(t)$. If $\mathbb{C}_r(t)$ is smaller, the tensor recomputing would be chosen for evicting and regenerating t. Otherwise, the swapping would be applied.

4.3 Tensor Memory Allocation

Assume that there are N memory allocation requests in the task queue. The i-th request can be described as a quadruple $r_i = (s_i, e_i, m_i, a_i)$ $(i = 1, 2, \cdots, N)$, where s_i and e_i represent the allocation and deallocation moment of the i-th request respectively, and m_i is its memory size, and a_i is the undetermined starting memory address for r_i. Each memory access request involves memory allocation denoted as (Allocation, s_i, m_i) and memory deallocation denoted as (Deallocation, e_i, m_i), which are arranged in the chronological order. If two requests r_i and r_j satisfy either $s_i \le e_j$ or $s_j \le e_i$, they conflict with each other in time. If r_i and r_j meet the condition $a_i \le a_j + m_j$ or $a_j \le a_i + m_i$, they have the space conflict. The design of TMA is to minimize the maximal memory footprint under the condition that there is no space conflict for any two requests which conflict in time. Our proposed TMA includes three steps:

Step 1: Before the training starts, six efficient memory management rules are established. As shown in Figure 6, they are (1) ***Alloc***(Panel ❶); (2) ***Alloc & Division***(Panel ❷); (3) ***Swell***(Panel ❸); (4) ***Free***(Panel ❹); (5) ***Free & Merge***(Panel ❺); (6) ***Overwrite & Free***(Panel ❻).

Step 2: During the first several iterations, the tensor access information is collected, e.g., memory request, data dependency, etc. The information embodies the invariable memory access characteristics on DCG even though DCG is built dynamically.

Step 3: Based on the collected information, the tensor access process is simulated, and six rules are applied to the tensor access simulation for rearranging of tensor positions. The simulation results would lead to the optimal memory chunk address for each tensor.

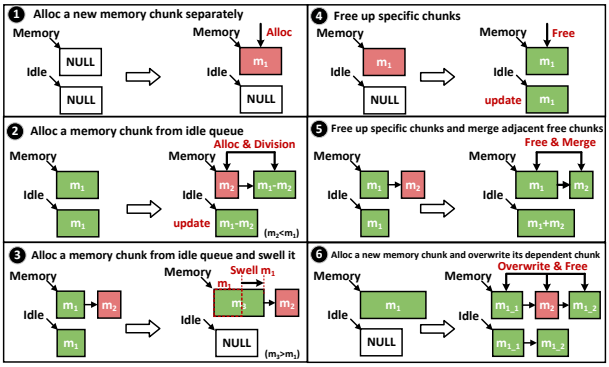


Figure 6: Six Efficient Memory Management Rules.

Figure 7 shows an example to illustrate how TMA applies six rules to a memory access simulation. Using the collected memory request and data dependency, we can obtain a memory access sequence, i.e., Alloc/Free Sequence as shown in Figure 7. For each request in the sequence, one of the six rules is selected to perform it. A linked-list *Memory List* records the memory usage, where each block represents a memory chunk for some tensor, and m_i means the memory chunk size. The orange and green colors represent *alloc* and *free* respectively. The linked-list *IdleList* records the free memory chuck. During the simulation, the relative positions of memory chunks are recorded by a topology graph. Based on the topology

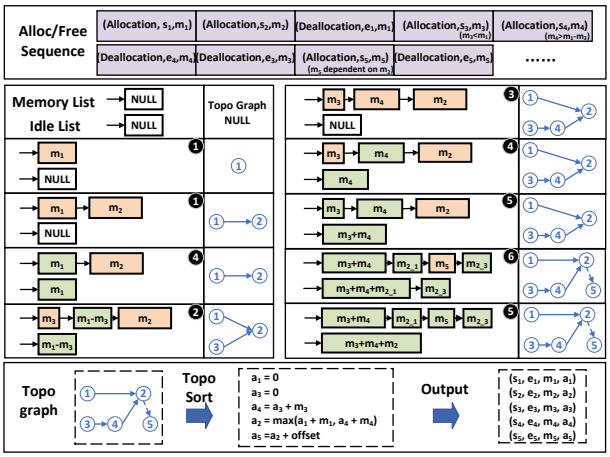


Figure 7: Memory Access Simulation. The memory list records the memory usage, and the idle list records which memory chunk is free (sorted by size).

graph, we define $Front_i$ $(1 \le i \le N)$, which is a array contains all frontend non-empty chunk indices of the i-th request at some moment. The starting address of the i-th request r_i is determined based on $front_i$.

As Figure 7 shown, there are 9 memory requests in the Alloc/Free sequence, and each request is implemented in a simulation stage. At the beginning of the memory access simulation, the memory list is empty. At the first two stages, the requests (*Allocation*, m_1) and (*Allocation*, m_2) are implemented based on Rule ❶. As m_1 is in front of m_2, we get $front_2 = \{1\}$. Next, the request (*Deallocation*, m_1) is implemented by Rule ❹ at the third stage, and (*Allocation*, m_3) is simulated using Rule ❷ at the fourth stage. At the same time, the set $front_2$ updates to $front_2 = \{1, 3\}$. At the fifth stage, as the idle chunk is small than m_4, Rule ❸ is selected to implement the request (*Allocation*, m_4). $front_2$ is changed into $front_2 = \{1, 4\}$, and $front_4$ is built as $front_4 = \{3\}$. Similarly, the remained requests are implemented stage by stage. The simulation process would generate the final topology graph and the sets $front_i$. Based on $front_i$, the starting memory address for the i-th request can be determined. Specifically, if $front_i = \emptyset$, set $a_i = 0$. Otherwise, let $a_i = \max_{j \in front_i}\{a_j + m_j\}$.

In addition, we can prove that the rule-based simulation above results in the optimal memory allocation which could minimize the maximal memory footprint under the condition that there is no space conflict for any two requests which conflict in time.

4.4 Overhead Analysis

In the following, we analyze the overhead of MegTaiChi, which includes the costs of DTP, DTE and TMA on DCG.

First, the cost of DTP mainly involves the runtime overhead of individual tensor partition inference and the overhead of data redistribution. On the one hand, as there are only 4 candidate solutions for the optimization problem (1), the overhead for tensor partition inference can be negligible compared to the runtime of each iteration. On the other hand, the overhead of data redistribution mainly is taken for changing the partition pattern of the current operator's weights from one kind to another. For example, assume that the

Table 2: Data redistribution cost for the tensor partition pattern switching. p is the number of devices, and $|T|$ represents the size of a global tensor T.

Original Pattern	Current Pattern inferred by DTP			
	P1	P2	P3	P4
P1	0	0	0	0
P2	$O((p-1)\cdot\|T\|)$	0	0	$O\left(\frac{p-1}{p}\|T\|\right)$
P3	$O((p-1)\cdot\|T\|)$	0	0	$O\left(\frac{p-1}{p}\|T\|\right)$
P4	$O((p-1)\cdot\|T\|)$	$O\left(\frac{p-1}{p}\|T\|\right)$	$O\left(\frac{p-1}{p}\|T\|\right)$	0

tensor partition pattern of OP_i is P3 at the last iteration, and DTP infers the tensor partition pattern of OP_i as P1 at the current iteration. MegTaiChi needs to perform an Allgather communication to change the partition pattern of the relevant weights for OP_i from P3 to P1 before executing OP_i. Denote the size of the global tensor T (weights) for OP_i as $|T|$. The required overhead for the data redistribution is about $O((p-1)\cdot|T|)$. Table 2 shows all the possible scenarios of the cost for data redistribution. By the way, the tensor partition pattern of an operator switches from P1 to any others, which involves no communication.

Next, the overhead of DTE includes two parts: the cost for searching the tensors to evict, and the cost for tracking and maintaining meta-data for each tensor in the candidate set. For searching tensors to evict, DTE has to traverse all the candidate tensors once to score each tensor and find the optimal tensor with the minimum score, which is time-consuming. For tracking the meta-data of each tensor, DTE needs to update and maintain the metadata of the residual tensors in the evicted candidate set, whose overhead can be hidden during the training process.

Finally, the overhead of TMA can be negligible, because TMA is executed only once after the warm up phase, and its main cost is for sorting the topology graph, whose time complexity is $O(N\log(N))$ for N memory allocation requests.

5 EVALUATION

5.1 Platform and Workload

Platform: Our experiment is performed on a modest cloud cluster, and the configuration is shown in Table 3.

Table 3: The Platform Configuration.

Number of Nodes (Machines)	32
Number of GPUs	256
GPU	Nvidia RTX 2080Ti (PCIe)x4 (16 GiB)
CPU	Intel(R) Xeon(R) E5-2650 v3 (32 GiB×8)
CPU-GPU Interconnect	PCI-Express Gen3x16 (16 GB/s)
GPU-GPU Interconnect	PCI-Express Gen3x16 (16 GB/s)
System Interconnect	25Gb Ethernet x 2 (6GB/s)

The experiment is running on Ubuntu 18.04. CUDA Toolkit version is 10.0, and cuDNN is 7.3.1. MegTaiChi is developed based on MegEngine 1.7 which is an open source imperative DL framework [1]. The evaluation focuses on the comparison of MegTaiChi with the state-of-the-art works, i.e., Capuchin [22], DTR [18] and Sublinear [6]. Capuchin is the state-of-the-art work proposing the swapping for tensor rematerialization, and DTR and Sublinear are the state-of-the-art works developing the recomputing. As Capuchin is not open source, we implement it as Meg-Capuchin using MegEngine. Meg-Capuchin can not run on multiple GPUs because its design applies only to single GPU. DTR and Sublinear are open source. And the original DTR is denoted as Pytorch-DTR which also can not run on multiple GPUs. In order to compare DTE with DTR on multiple GPUs, we develop Meg-DTR based on MegEngine. Table 4 presents all the methods in the evaluation.

In MegTaiChi, DTP, DTE and TMA can work individually and coordinate with each other. DTP and DTE can support both static and dynamic networks. If the neural network architecture is fixed, TMA can manage the memory allocation in fine-grain.

Workload: We evaluate MegTaiChi on the DNNs shown in Table 5, including nine representative DNNs with fixed architectures and two DNN with unfixed architecture changing during the training.

Table 4: The Methods using in the evaluation.

Symbol	Definition
Meg-Capuchin	Our implementation of Capuchin [22] on MegEngine.
Pytorch-DTR	Open source version of DTR [18] on Pytorch.
Meg-DTR	Our implementation of DTR [18] on MegEngine.
Sublinear	Open source version of Sublinear [6] on Pytorch.
MegEngine-dir	Directly apply MegEngine to training DNN.
DTP	Dynamic Tensor Partition.
DTE	Dynamic Tensor Evicting.
TMA	Tensor Memory Allocation.
DP	Date Parallelism Strategy.
MP	Model Parallelism Strategy.
DP+MP	Empirical strategy using DP and MP for different layers.

Table 5: DNN Models for Evaluation.

Model	Type	Architecture	Num. of Param.
ShuffleNet[34]	CNN	fixed	2.3M
VGG-16[12]	CNN	fixed	138M
ResNet-50[12]	CNN	fixed	25.6M
ResNet-101[12]	CNN	fixed	44.5M
ResNet-152[12]	CNN	fixed	60.2M
Bert-base[7]	Transformer	fixed	110M
SPOS[11]	CNN	unfixed	3.4M
Global Local (GL)-base	CNN	fixed	100M
Global Local (GL)-large	CNN	fixed	1.5B
GPT[5]	Transformer	fixed	1.3, 2.6, 6.7, 12(B)
MoE[19]	MoE	unfixed	0.35, 1.3, 2.4, 8(B)

* Both GPT and MoE have four sizes, because the two models become larger with the number of GPUs increasing from 1 to 8 in the weak scalability tests.

5.2 Evaluation of DTP

For the evaluation of DTP, we select four DNN models, as shown in Figure 9. Compared with DP, the contribution of DTP on performance improvement is slight for ResNet-50 model, while DTP achieves 1.3×, 1.4× and 1.5× performance speedups for VGG-16, GL-base and GL-large models respectively. The main reason is that ResNet-50 is a convolution-dominated network where each convolution layer is usually small and prefers data parallelism (essentially Pattern 1). However, VGG-16, GL-base and GL-large models all contain many fully-connected layers. We find DTP adaptively applies Pattern 1 for convolution layers, Pattern 3 and Pattern 4 for fully-connected layers, which leads to the performance improvement.

5.3 Evaluations of DTE and TMA

For the evaluations of DTE and TMA, the test results on single GPU and multiple GPUs are presented separately. In this section, the

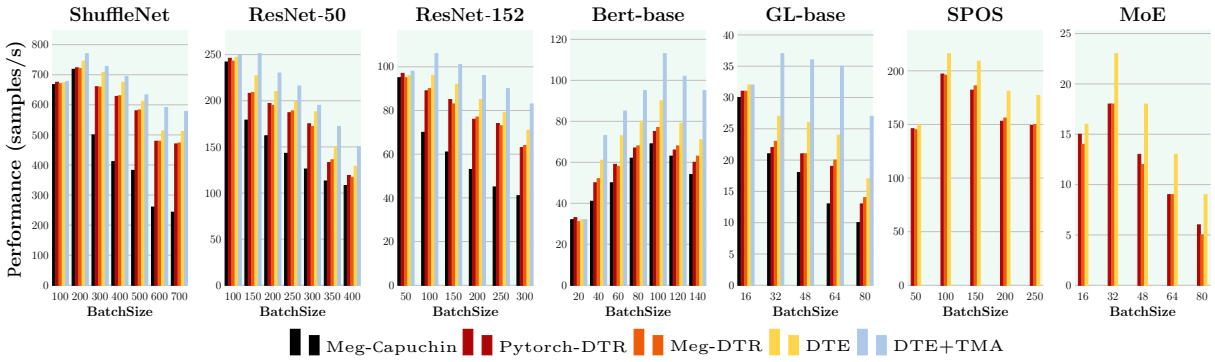


Figure 8: Evaluations on a Single GPU. Meg-Capuchin, Pytorch-DTR and Meg-DTR are three baselines.

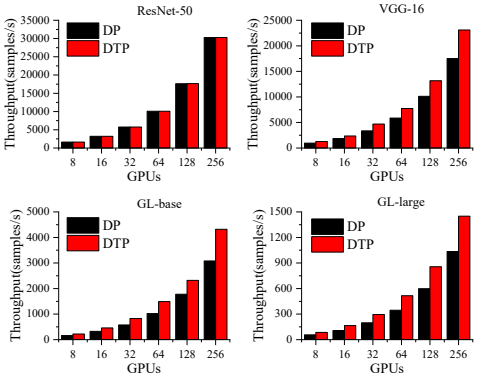


Figure 9: Evaluation of DTP.

tests on multiple GPUs do not involve DTP, while DP is applied for the tensor partitions of all the layers.

5.3.1 Test Results on Single GPU. We compare the throughputs of Meg-Capuchin, Pytorch-DTR, Meg-DTR, DTE and TMA on the training of 5 DNN models. Before the comparison, we evaluate the performance of two deep learning frameworks Pytorch and Megengine. By running 10 iterations of the ResNet-50 training, two frameworks achieve almost the same performance with small gap less than 10 ms, which help us focus on the comparison of different methods.

As shown in Figure 8, DTE achieves about 1.5× and 1.23× performance speedups on average compared with Capuchin and Pytorch-DTR respectively. Further, by cooperating DTE with TMA, the average throughput improvements increase to 1.6× and 1.5× faster than Capuchin and Pytorch-DTR respectively. We have three important observations from the experiment results.

First, for the training of Bert-base model, the throughputs of different methods are increasing as the batch sizes increases from 20 to 100, because the free memory is relatively abundant during this process. With the continuous increase of batch sizes from 100 to 140, the throughputs decrease monotonically. This is because the tensor swapping/recomputing operations occur frequently when the free memory becomes insufficient, which affects the execution efficiency. The training of ShuffleNet model follows the similar trend. However, for ResNet-50, ResNet-152 and GL-base, the smallest batch sizes results in insufficient free memory. Hence, the throughputs of different methods decrease with the increase of batch sizes.

Table 6: Recomputing and swapping costs for a conv-BN-relu block on 1 GPU and 8 GPUs.

Batch Sizes	Recomputing Cost (ms)			Recomputing Frequency	Swapping Cost (ms)
	Convolution layer	BN layer	Relu layer		
32 (24.5 M)	(0.12, 0.12)	(0.19, 0.20)	(0.10, 0.11)	(0, 0)	(2.18, 7.98)
64 (49 M)	(0.22, 0.22)	(0.34, 0.36)	(0.20, 0.21)	(0, 0)	(4.36, 43.11)
128 (98 M)	(0.41, 0.41)	(0.64, 0.67)	(0.39, 0.39)	(2, 2)	(8.71, 26.22)
256 (196 M)	(0.81, 0.81)	(1.23, 1.24)	(0.77, 0.78)	(2, 3)	(17.41, 52.26)

Here, (a, b) shows test results a and b on 1 GPU and 8 GPUs respectively.

Second, as shown in Table 6, the execution time of the swapping is dozens of the cost for recomputing on our platform. As Capuchin does not make full use of the runtime meta-information of the tensors to determine the recomputing plan, Meg-Capuchin is slower than Pytorch-DTR and Meg-DTR.

Third, compared with Pytorch-DTR and Meg-DTR, DTE achieves about 1.24× and 1.22× performance speedup on average respectively. This mainly owes to the design of DTE which takes the impact of memory fragmentation into account. Reducing the generation of memory fragmentation can decrease the frequency of tensor evicting operations, resulting in the performance improvement. Further, by coordinating DTE and TMA, i.e., DTE+TMA, the memory usage follows a fine-grained allocation, and the higher performance speedup can be achieved.

5.3.2 Test Results on Multiple GPUs. Figure 10 shows the test results on 8 GPUs. Compared with Meg-DTR, DTE achieves 1.6× speedup on average. Further, coordinating DTE and TMA can obtain 1.4× performance improvement compared with only using DTE. This is because DTE+TMA achieves a fine-grained memory allocation so that the memory fragmentation is further reduced and the evicting operation becomes less frequent. It should be noted that the throughputs of different methods follow the similar change trends with the increase of batch sizes whenever using a single GPU or 8 GPUs, while the throughput on 8 GPUs is obviously higher than that on a single one.

5.4 Evaluation on Dynamic Network Models

For dynamic models, the execution sequence of operators is not fixed. In the related works, only DTR [18] can support the training of dynamic neural networks currently. Hence, the evaluation on dynamic models focuses on the comparison of DTE with DTR and Meg-DTR.

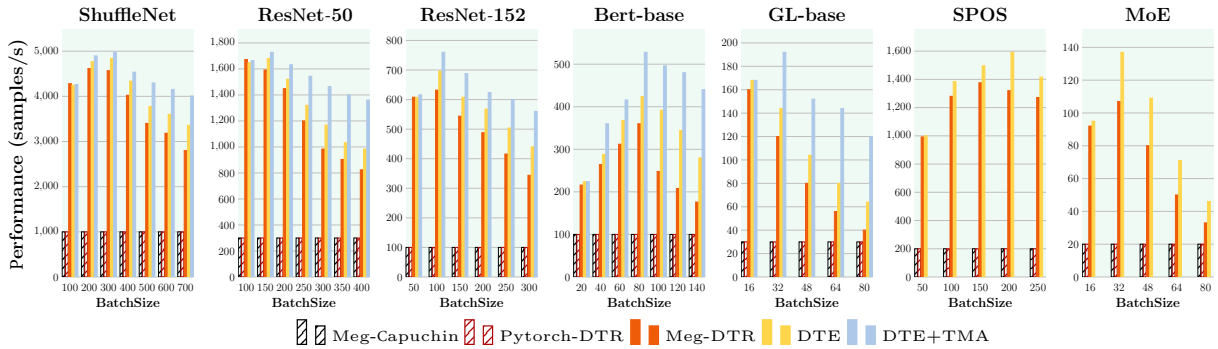


Figure 10: Evaluations on 8 GPUs. The bars with slashes mean that the relevant methods can not run on multiple GPUs, e.g., Meg-Capuchin and Pytorch-DTR.

Figure 8 and Figure 10 show the performances of different methods for the training of two dynamic models, i.e., SPOS and MoE. Compared with Pytorch-DTR and Meg-DTR, DTE achieves 1.3× and 1.4× performance speedups on average respectively. Further, the improvement advantage of DTE over 8 GPUs is higher than that on a single GPU. It is worth mentioning that, DTR [18] just consider tensor rematerialization on single GPU. However, MegTaiChi extends dynamic tensor rematerialization to multiple GPUs and reduces memory fragmentation.

5.5 On Memory Usage

Figure 11 shows the memory usage at any moment of each training iteration using TMA and Pytorch-DTR. In each panel of Figure 11, X-axis and Y-axis represent the memory address and the time sequence respectively. Along X-axis, the leftmost grey region is for storing the model parameters and gradients (the first region m_1 mentioned in Section 4.2), and the color region is for tensors taking part in the current computation (the second region m_2 mentioned in Section 4.2). It is clear that, compared with Pytorch-DTR, TMA results in less memory fragments. Due to this fact, the peak memory usage is reduced by 11% for ResNet-50 and 10.5% for GL-base respectively, which is a significant achievement for the memory optimization based on DCG [6, 18, 22].

With the continuous increase of batch sizes, we find that OOM would occur even though DTR is applied. As shown in Table 7, when the batch sizes increases to 400 on a single GPU, the training of ResNet-50 model has to leverage the defragmentation procedure to enable the execution continue. To generate a new and complete free memory space which should be sufficient for the current execution, the defragmentation procedure involves the reallocation of all the data in the on-device memory and puts all the memory fragments together, which is time consuming. Table 7 presents that the cost of one defragmentation procedure is beyond one thousand milliseconds on average, which affects the execution overhead seriously. From Table 7, it is clear that Pytorch-DTR still has to call the defragmentation procedure multiple times per 10 iterations for the training of different models with large batch sizes. However, in the same cases, TMA successfully avoids the defragmentation. This fact proves again that TMA could effectively reduce the generation of memory fragments due to the fine-grained memory allocation.

Table 7: Defragmentation costs.

Model	Batch Sizes	Defragmentation		
		Average execution cost of one defragmentation	Frequency per 10 iterations	
			Pytorch-DTR	TMA
ResNet-50	400	1367 ms	19	0
ResNet-101	300	1605 ms	6	0
ResNet-152	300	1471 ms	9	0
GL-base	90	1048 ms	10	0

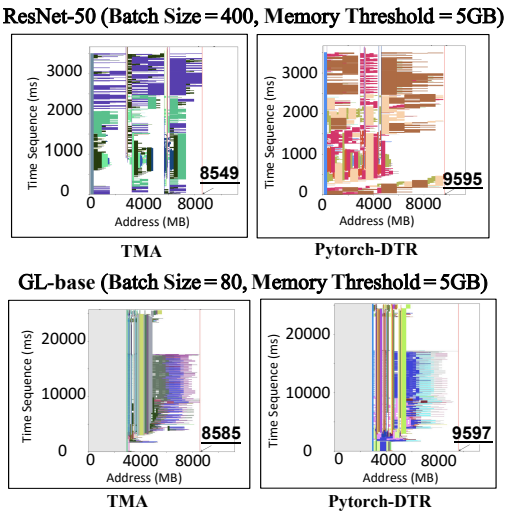


Figure 11: Memory usage within each iteration. The grey regions for ResNet-50 are imperceptible since the memory size storing model parameters and gradients is only about 90MB.

5.6 Evaluation of MegTaiChi

In this section, we evaluate MegTaiChi's overhead, we compare MegTaiChi with Sublinear [6] which is famous for its memory optimization to support the large batch training on a single GPU, and then we apply MegTaiChi to the training of a large-scale face recognition mode.

5.6.1 Runtime Overhead of MegTaiChi. As Figure 12 shown, the runtime overhead of MegTaiChi is less than 5% of the runtime of each iteration. From Figure 12, we obtain three observations.

First, the overhead of MegTaiChi on a single GPU is smaller than that on multiple GPUs, because DTP is unnecessary for the training on a single GPU.

Second, compared with the networks such as ResNet-50, ResNet-152 and SPOS, GL-base network contains more fully-connected layers and less convolution layers, which involves more communication and memory usage costs. Therefore, the overhead of MegTaiChi for the training of GL-base model increases to about 1.5 times of the costs for the other networks.

Third, for the training of SPOS network, although SPOS is dynamic, the overhead of MegTaiChi introduced among all workloads are less than 3%, and the average is 2.68%.

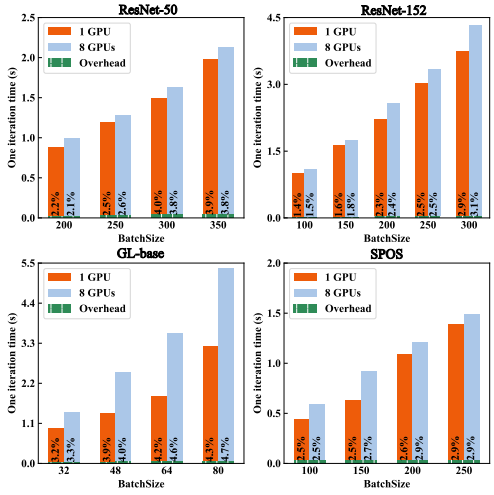


Figure 12: Runtime overhead of MegTaiChi.

5.6.2 MegTaiChi vs. Sublinear. Figure 13 shows a comparison of MegTaiChi with Sublinear in the training of ResNet-50, where the memory threshold m^* increases from 4G to 6G (the m^* mentioned in Section 4.2). We conclude three important observations from the experiment results. First, for the same batch size, increasing m^* enables the performance of MegTaiChi gradually become closer to Sublinear. When m^* is set as 6G, MegTaiChi is slightly slower than Sublinear, while the gap is less than 0.04s for each iteration. Second, when the batch sizes are 100, 150 and 200 respectively, it is obvious that MegTaiChi consumes less memory than Sublinear. Third, for the training of ResNet-50 on a single GPU, the maximum batch size for Sublinear is 200, while MegTaiChi supports larger batch sizes. As Figure 13 shown, the memory consumption for MegTaiChi with $bs = 300$ is still less that for Sublinear with $bs = 200$.

Furthermore, Table 8 presents that MegTaiChi supports larger maximum batch sizes compared with the others. MegTaiChi achieves the maximum batch sizes by up to 5× and 2.4× compared with MegEngine-dir and Sublinear respectively on average.

Table 8: Maximum batch sizes supported by different methods.

Model	MegEngine-dir	Sublinear [6]	MegTaiChi
ShuffleNet	200	550	950
ResNet-50	110	200	430
ResNet-152	50	130	300
GL-base	16	45	90
Bert-base	22	45	140
SPOS	50	N/A	350

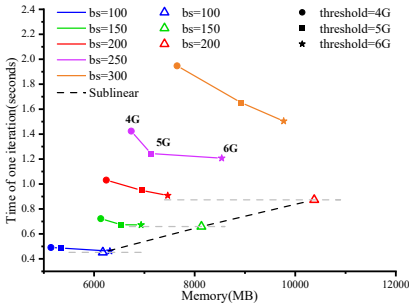


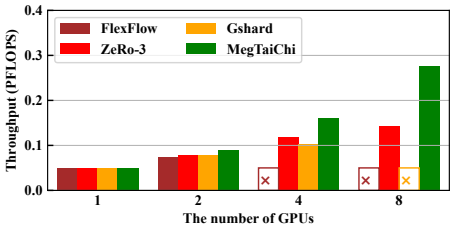
Figure 13: MegTaiChi vs. Sublinear in the training of ResNet-50 model on a single GPU.

5.6.3 MegTaiChi vs. FlexFlow, Gshard and ZeRo-3. For the comparison of MegTaiChi with FlexFlow, Gshard and ZeRo-3, the weak scalability of different modules is evaluated by increasing the batch size and model size along with the number of GPUs. As the model sizes are different for different numbers of GPUs, it seems not reasonable to use throughput as the metric, e.g., tokens per second. Hence, we choose the aggregated peta floating-point operations per second (PFLOPS) of GPUs as the evaluation metric by following the suggestion from recent works [28]. As shown in Figure 14, compared with FlexFlow, Gshard and ZeRo-3, MegTaiChi achieves 1.2×, 1.8× and 1.5× performance speedups respectively on average.

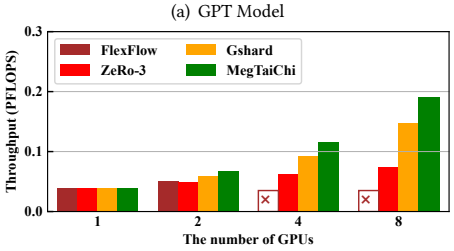
With the increase of the batch size and model size, the training process needs more additional memory spaces for the data movement between GPUs. From Figure 14, it is clear that FlexFlow and Gshard fail to train the GPT model in the weak scaling test over 8 GPUs. In fact, FlexFlow and GShard try to exploit the parallelism of multi-dimensional tensor partition on SCG. However, MegTaiChi abstracts partition patterns to immediately generate the memory optimization plan on DCG, leading to the better behavior on the memory usage.

ZeRo-3 manages the memory usage by manually tuning tensor rematerialization and tensor partition. Hence, it is able to train the networks with larger batch size and model size. But, ZeRo-3 does not focus on the performance optimization of tensor rematerialization and tensor partition behaviors. Specifically, ZeRo-3 has to communicate all the gradients between GPUs. When the gradients are larger than activations, its performance degenerates. However, Megtaichi automatically combines tensor partition with tensor rematerialization to determine the near-optimal memory management plan on DCG, which could achieve higher performance.

5.6.4 Training of Large-scale Face Recognition Models. The final experiment evaluates MegTaiChi with the training of large-scale face recognition models. The models are chosen from Global Local family which includes GL-base and GL-large. In real applications, GL-large supports more than one million face classification. The architectures of GL-base and GL-large interleave convolution layers and fully-connected layers. From Figure 15, we have four important observations. First, DTP achieves 1.3× performance speedup on average compared with the empirical strategy DP+MP. It is because DTP dynamically guides more reasonable tensor partition by balancing the execution cost and memory usage at the runtime. Second, the coordinated design of DTP and DTE achieves 1.2× performance speedup on average compared with DTE. This mainly



(a) GPT Model



(b) MoE Model

Figure 14: Weak scalability tests. The notation × presents that the out of memory error occurs.

owes to the fact that DTE further achieves larger batch sizes within the memory-limit constraint. Third, on average, TMA enables the coordinated design of DTP and DTE further achieve 1.2× speedup due to the fine-gained memory allocation which reduces the memory fragments and avoids the defragmentation procedure. Fourth, by combining DTP, DTE and TMA together, MegTaiChi achieves 1.87× speedup on average compared with the empirical DP+MP.

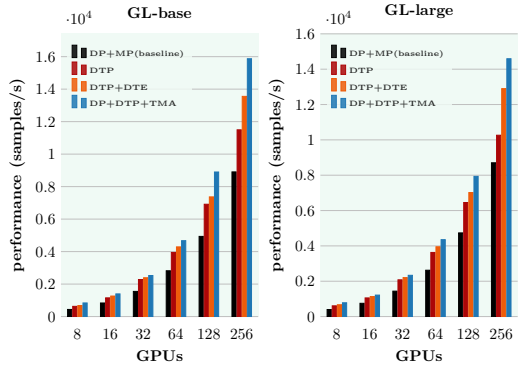


Figure 15: Training of Large-scale Face Recognition Models.

6 RELATED WORKS

Tensor Partition: To facilitate deep learning model training, the current deep learning frameworks, e.g., TensorFlow [2] and PyTorch [21], provide well-supported data parallelism and vanilla model parallelism by explicitly assigning operations to specific devices. Mesh-TensorFlow [25] designs a special language for rewriting DNN models to achieve the distributed training. Tofu [31] requires developers to specify the tensor partition for operators using a description language called TDL. Megtron [28] and DeepSpeed [23] successfully couple the implementation of tensor model parallelism with model programming. GShard [19] uses parallel annotations to infer the partition of the remained tensors. However, these major related works are based on SCG, and their tensor partitions are built empirically before the execution. To the best of our knowledge, MegTaiChi proposes a *dynamic tensor partition* to adjust the partition for each tensor on DCG during the training process. The design of DTP takes the runtime information into account and provides an opportunity for effectively coordinating the tensor partition and tensor rematerialization techniques.

Tensor Rematerialization: The majority works perform memory optimization based on computation graph, including two categories, i.e., swapping and recomputation. For the swapping, vDNN [24] and SuperNeurons [30] propose to swap the data out to the CPU in forward phase and prefetch it at backward phase, which enables the training of large models under restricted memory budgets. SwapAdvisor [13] designs a genetic algorithm to automatically generate the swapping plan based on SCG. Further, Capuchin [22] represents the state-of-the-art work developing the swapping on both SCG and DCG. It collects the information of tensor access at runtime for memory management. On the other hand, for the recomputing, Sublinear [6] proposes to release some tensors in the forward propagation and recompute them during backward propagation, which can train an N-layer linear feed-forward network on $O(\sqrt{N})$ memory budget with $O(N \log(N))$ extra recomputation cost. Sublinear well supports the training with large batch size under limited memory constraint. Checkmate [14] analyzes the training on SCG and constructs an integer linear programming to find the optimal evicted tensors. To the best of our knowledge, DTR [18] is the state-of-the-art work developing the recomputing on DCG. DTR applies heuristics to guide its eviction choices during the execution. However, the related works just focus on the tensor eviction with cheap regeneration costs, while ignore the variation among the same size tensors stored at different spatial positions of memory. Thus, They can not fully extract the memory reduction opportunity. In this work, MegTaiChi proposes a *dynamic tensor evicting* which leverages the variation among tensors at different memory locations to reduce the memory fragmentation. Further, a *tensor memory allocation* is designed to achieve the fine-grained memory management for each tensor.

7 CONCLUSION

This work proposes MegTaiChi, a dynamic tensor-based memory management optimization module for the DNN training, which achieves an efficient coordination of tensor partition and tensor rematerialization. Specifically, DTP, DTE and TMA are designed for the heuristic, adaptive and fine-grained memory management. The experimental results confirm the high performance of MegTaiChi.

ACKNOWLEDGMENTS

The authors would like to thank all anonymous reviewers for their valuable comments and helpful suggestions. The work is supported by the National Key Research and Development Program of China under Grant No. 2018AAA0103302, and National Natural Science Foundation of China, under Grant No. (62172391, 61972377, 62032023, T2125013).

REFERENCES

[1] 2022. MegEngine is a fast, scalable and easy-to-use deep learning framework, with auto-differentiation. https://github.com/MegEngine/MegEngine.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[3] Shriram B, Anshuj Garg, and Purushottam Kulkarni. 2019. Dynamic Memory Management for GPU-Based Training of Deep Neural Networks. 200–209. https://doi.org/10.1109/IPDPS.2019.00030

[4] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. 2019. Qsparse-local-SGD: Distributed SGD with quantization, sparsification, and local computations. *arXiv preprint arXiv:1906.02367* (2019).

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[8] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. 2019. Channel and Filter Parallelism for Large-Scale CNN Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 10, 20 pages. https://doi.org/10.1145/3295500.3356207

[9] Amir Gholami, Ariful Azad, Kurt Keutzer, and Aydin Buluç. 2017. Integrated Model and Data Parallelism in Training Neural Networks. *CoRR* abs/1712.04432 (2017). arXiv:1712.04432 http://arxiv.org/abs/1712.04432

[10] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 51–62.

[11] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. 2019. Single Path One-Shot Neural Architecture Search with Uniform Sampling. *CoRR* abs/1904.00420 (2019). arXiv:1904.00420 http://arxiv.org/abs/1904.00420

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[13] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. https://doi.org/10.1145/3373376.3378530

[14] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 497–511. https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf

[15] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Langshi chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2021. Whale: Scaling Deep Learning Model Training to the Trillions. arXiv:2011.09208 [cs.DC]

[16] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).

[17] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks.. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 1–13. https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf

[18] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. arXiv:2006.09616 [cs.LG]

[19] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs.CL]

[20] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 1–15.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[22] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. https://doi.org/10.1145/3373376.3378505

[23] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[24] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783721

[25] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/3a37abdeefe1dab1b30f7c5c7e581b93-Paper.pdf

[26] Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. 2019. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772* (2019).

[27] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[28] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[29] Mohamed Wahib, Haoyu Zhang, Truong Thao Nguyen, Aleksandr Drozd, Jens Domke, Lingqi Zhang, Ryousei Takano, and Satoshi Matsuoka. 2020. Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA. arXiv:2008.11421 [cs.DC]

[30] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3178487.3178491

[31] Minjie Wang, Chien chin Huang, and Jinyang Li. 2018. Unifying Data, Model and Hybrid Parallelism in Deep Learning via Tensor Tiling. arXiv:1805.04170 [cs.DC]

[32] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models Using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/3302424.3303953

[33] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. *Proceedings of the Fourteenth EuroSys Conference 2019* (Mar 2019). https://doi.org/10.1145/3302424.3303953

[34] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *CoRR* abs/1707.01083 (2017). arXiv:1707.01083 http://arxiv.org/abs/1707.01083